Forth (programming language)

Forth is an imperative stack-based computer programming language and environment originally designed by Charles H. "Chuck" Moore. Language features include structured programming, reflection (the ability to examine and modify program structure during execution), concatenative programming (functions are composed with juxtaposition) and extensibility (the programmer can create new commands). Although not an acronym, the language's name is its early years was often spelled in all capital letters as *FORTH*, but *Forth* is more common.

A procedural programming language without type checking, Forth provides interactive execution of commands, allowing introspection of a live system, while still retaining the ability to compile sequences of commands for later execution. For much of Forth's existence, the standard technique was to compile to threaded code, which can be interpreted faster than bytecode, but there are modern implementations that generate optimized machine code like other language compilers.

Forth is used in the Open Firmware boot loader, in space applications[1] such as the Philae spacecraft,[2][3] and in other embedded systems which involve interaction with hardware. The bestselling 1986 computer game *Starflight*, from Electronic Arts, was written with a custom Forth.[4] The free software Gforth implementation is actively maintained, as are several commercially supported systems.

Moore later experimented with more minimal languages based on Forth concepts, including cmForth and colorForth.

Forth

Paradigm	Procedural, stack-oriented, reflective, concatenative
Designed by	Charles H. Moore
First appeared	1970
Typing discipline	typeless
Filename extensions	.fs, .fth, .forth
Major implementations	
SwiftForth (Forth, Inc.) Gforth (Free software) VFX Forth (MicroProcessor Engineering)	
Influenced by	
Burroughs large systems, Lisp, APL	
Influenced	
Factor, Joy, PostScript, RPL, REBOL	

Contents

Overview

Forth is a simple, yet extensible language; its modularity and extensibility permit writing significant programs.

A Forth environment combines the compiler with an interactive shell, where the user defines and runs subroutines called *words*. Words can be interactively defined, tested, redefined, and debugged without recompiling or restarting the whole program. All syntactic elements, including variables and basic operators, are defined as words. Ideally running the program has the same effect as manually re-entering the source.

The Forth philosophy emphasizes the use of small, simple words. Words for bigger tasks call upon many smaller words that each accomplish a distinct sub-task. A large Forth program is a hierarchy of words. These words, being distinct modules that communicate (pass data) implicitly via a stack mechanism, can be prototyped, built and tested independently. The highest level of Forth code may resemble an English-language description of the application. Forth has been called a "meta-application language": a language that can be used to create problem-oriented languages.[5]

Uses

Forth has a niche both in astronomical and space applications[6] as well as a history in general embedded systems. The Open Firmware boot ROMs used by Apple, IBM, Sun, and OLPC XO-1 contain a Forth environment.

Forth has often been used to bring up new hardware. For example, Forth was the first resident software on the new Intel 8086 chip in 1978 and MacFORTH was the first resident development system for the 128K Macintosh in 1984.[7]

Atari, Inc. used an elaborate animated demo written in Forth to showcase capabilities of the Atari 400 and 800 computers in department stores.[8] Two home computer games from Electronic Arts, published in the 1980s, were written in Forth: *Worms?* (1983)[9] and *Starflight* (1986).[4] The Canon Cat (1987) uses Forth for its system programming.

Rockwell produced single-chip microcomputers with resident Forth kernels, the R65F11 and R65F12. ASYST was a Forth expansion for measuring and controlling on PCs.[10]

History

Forth evolved from Charles H. Moore's personal programming system, which had been in continuous development since 1968.[7] Forth was first exposed to other programmers in the early 1970s, starting with Elizabeth Rather at the United States National Radio Astronomy Observatory (NRAO).[7] After their work at NRAO, Charles Moore and Elizabeth Rather formed FORTH, Inc. in 1973, refining and porting Forth systems to dozens of other platforms in the next decade.

Forth is so-named, because in 1968 "the file holding the interpreter was labeled FOURTH, for 4th (next) generation software, but the IBM 1130 operating system restricted file names to five characters."[11] Moore saw Forth as a successor to compile-link-go third-generation programming languages, or software for "fourth generation" hardware.

FORTH, Inc.'s microFORTH was developed for the Intel 8080, Motorola 6800, Zilog Z80, and RCA 1802 microprocessors, starting in 1976. MicroFORTH was later used by hobbyists to generate Forth systems for other architectures, such as the 6502 in 1978. Common practice was codified in the de facto standards FORTH-79[12] and FORTH-83[13] in the years 1979 and 1983, respectively. These standards were unified by ANSI in 1994, commonly referred to as ANS Forth.[14][15]

Forth became popular in the early 1980s[16] because it was well suited to the limited memory of microcomputers. The ease of implementing the language led to many implementations.[17] The British Jupiter ACE home computer has Forth in its ROM-resident operating system. Insoft GraFORTH is a version of Forth with graphics extensions for the Apple II.[18]

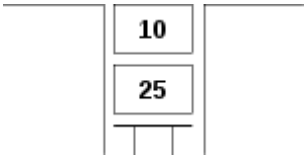
As of 2018, the source for the original 1130 version of FORTH has been recovered, and is now being updated to run on a restored or emulated 1130 system.[19]

Programmer's perspective

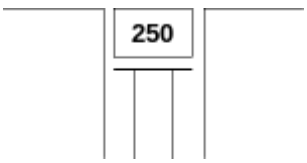
Forth relies on explicit use of a data stack and reverse Polish notation which is commonly used in calculators from Hewlett-Packard. In RPN, the operator is placed after its operands, as opposed to the more common infix notation where the operator is placed between its operands. Postfix notation makes the language easier to parse and extend; Forth's flexibility makes a static BNF grammar inappropriate, and it does not have a monolithic compiler. Extending the compiler only requires writing a new word, instead of modifying a grammar and changing the underlying implementation.

Using RPN, one can get the result of the mathematical expression `(25 * 10 + 50)` this way:

```
25 10 * 50 + CR .
300 ok
```

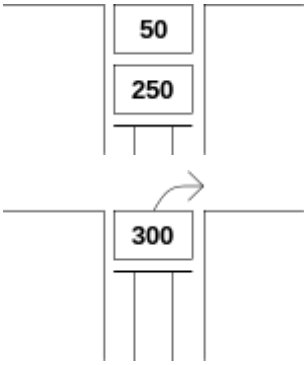


First the numbers 25 and 10 are put on the stack.



The word * takes the top two numbers from the stack, multiplies them, and puts the product back on the stack.

Then the number 50 is placed on the stack.



The word `+` adds the top two values, pushing the sum. `CR` (carriage return) starts the output on a new line. Finally, `.` prints the result. As everything has completed successfully, the Forth system prints `OK`.[20]

Even Forth's structural features are stack-based. For example:

```
: FLOOR5 ( n -- n' )   DUP 6 < IF DROP 5 ELSE 1 - THEN ;
```

The colon indicates the beginning of a new definition, in this case a new word (again, *word* is the term used for a subroutine) called `FLOOR5`. The text in parentheses is a comment, advising that this word expects a number on the stack and will return a possibly changed number (on the stack).

The subroutine uses the following commands: `DUP` duplicates the number on the stack; `6` pushes a 6 on top of the stack; `<` compares the top two numbers on the stack (6 and the `DUP`ed input), and replaces them with a true-or-false value; `IF` takes a true-or-false value and chooses to execute commands immediately after it or to skip to the `ELSE`; `DROP` discards the value on the stack; `5` pushes a 5 on top of the stack; and `THEN` ends the conditional.

The `FLOOR5` word is equivalent to this function written in the C programming language using the ternary operator '?:'

```
int floor5(int v) {
  return (v < 6) ? 5 : (v - 1);
}
```

This function is written more succinctly as:

```
: FLOOR5 ( n -- n' ) 1- 5 MAX ;
```

This can be run as follows:

```
1 FLOOR5 CR .
5 ok
8 FLOOR5 CR .
7 ok
```

First a number (1 or 8) is pushed onto the stack, `FLOOR5` is called, which pops the number again and pushes the result. `CR` moves the output to a new line (again, this is only here for readability). Finally, a call to `.` pops the result and prints.

Facilities

Forth's grammar has no official specification. Instead, it is defined by a simple algorithm. The interpreter reads a line of input from the user input device, which is then parsed for a word using spaces as a delimiter; some systems recognise additional whitespace characters. When the interpreter finds a word, it looks the word up in the *dictionary*. If the word is found, the interpreter executes the code associated with the word, and then returns to parse the rest of the input stream. If the word isn't found, the word is assumed to be a number and an attempt is made to convert it into a number and push it on the stack; if successful, the interpreter continues parsing the input stream. Otherwise, if both the lookup and the number conversion fail, the interpreter prints the word followed by an error message indicating that the word is not recognised, flushes the input stream, and waits for new user input.[21]

The definition of a new word is started with the word `:` (colon) and ends with the word `;` (semi-colon). For example,

```
: X DUP 1+ . . ;
```

will compile the word `X`, and makes the name findable in the dictionary. When executed by typing `10 X` at the console this will print `11 10`.[22]

Most Forth systems include an assembler that allows one to specify words using the processor's facilities at its lowest level. Mostly the assembler is tucked away in a separate namespace (*wordlist*) as relatively few users want to use it. Forth assemblers may use a reverse Polish syntax in which the parameters of an instruction precede the instruction, but designs vary widely and are specific to the Forth implementation. A typical reverse Polish assembler prepares the operands on the stack and have the mnemonic copy the whole instruction into memory as the last step. A Forth assembler is by nature a macro assembler, so that it is easy to define an alias for registers according to their role in the Forth system: e.g. "datastackpointer" for the register used as a stack pointer.[23]

Operating system, files, and multitasking

Most Forth systems run under a host operating system such as Microsoft Windows, Linux or a version of Unix and use the host operating system's file system for source and data files; the ANSI Forth Standard describes the words used for I/O. All modern Forth systems use normal text files for source, even if they are embedded. An embedded system with a resident compiler gets its source via a serial line.

Classic Forth systems traditionally use neither operating system nor file system. Instead of storing code in files, source code is stored in disk blocks written to physical disk addresses. The word `BLOCK` is employed to translate the number of a 1K-sized block of disk space into the address of a buffer containing the data, which is managed automatically by the Forth system. Block use has become rare since the mid-1990s. In a hosted system those blocks too are allocated in a normal file in any case.

Multitasking, most commonly cooperative round-robin scheduling, is normally available (although multitasking words and support are not covered by the ANSI Forth Standard). The word `PAUSE` is used to save the current task's execution context, to locate the next task, and restore its execution context. Each task has its own stacks, private copies of some control variables and a scratch area. Swapping tasks is simple and efficient; as a result, Forth multitaskers are available even on very simple microcontrollers, such as the Intel 8051, Atmel AVR, and TI MSP430.[24]

Other non-standard facilities include a mechanism for issuing calls to the host OS or windowing systems, and many provide extensions that employ the scheduling provided by the operating system. Typically they have a larger and different set of words from the stand-alone Forth's `PAUSE` word for task creation, suspension, destruction and modification of priority.

Self-compilation and cross compilation

A full-featured Forth system with all source code will compile itself, a technique commonly called meta-compilation or self-hosting, by Forth programmers (although the term doesn't exactly match meta-compilation as it is normally defined). The usual method is to redefine the handful of words that place compiled bits into memory. The compiler's words use specially named versions of fetch and store that can be redirected to a buffer area in memory. The buffer area simulates or accesses a memory area beginning at a different address than the code buffer. Such compilers define words to access both the target computer's memory, and the host (compiling) computer's memory.[25]

After the fetch and store operations are redefined for the code space, the compiler, assembler, etc. are recompiled using the new definitions of fetch and store. This effectively reuses all the code of the compiler and interpreter. Then, the Forth system's code is compiled, but this version is stored in the buffer. The buffer in memory is written to disk, and ways are provided to load it temporarily into memory for testing. When the new version appears to work, it is written over the previous version.

Numerous variations of such compilers exist for different environments. For embedded systems, the code may instead be written to another computer, a technique known as cross compilation, over a serial port or even a single TTL bit, while keeping the word names and other non-executing parts of the dictionary in the original compiling computer. The minimum definitions for such a Forth compiler are the words that fetch and store a byte, and the word that commands a Forth word to be executed. Often the most time-consuming part of writing a remote port is constructing the initial program to implement fetch, store and execute, but many modern microprocessors have integrated debugging features (such as the Motorola CPU32) that eliminate this task.[26]

Structure of the language

The basic data structure of Forth is the "dictionary" which maps "words" to executable code or named data structures. The dictionary is laid out in memory as a tree of linked lists with the links proceeding from the latest (most recently) defined word to the oldest, until a sentinel value, usually a NULL pointer, is found. A context switch causes a list search to start at a different leaf. A linked list search continues as the branch merges into the main trunk leading eventually back to the sentinel, the root. There can be several dictionaries. In rare cases such as meta-compilation a dictionary might be isolated and stand-alone. The effect resembles that of nesting namespaces and can overload keywords depending on the context.

A defined word generally consists of *head* and *body* with the head consisting of the *name field* (NF) and the *link field* (LF), and body consisting of the *code field* (CF) and the *parameter field* (PF).

Head and body of a dictionary entry are treated separately because they may not be contiguous. For example, when a Forth program is recompiled for a new platform, the head may remain on the compiling computer, while the body goes to the new platform. In some environments (such as embedded systems) the heads occupy memory unnecessarily. However, some cross-compilers may put heads in the target if the target itself is expected to support an interactive Forth.[27]

Dictionary entry

The exact format of a dictionary entry is not prescribed, and implementations vary. However, certain components are almost always present, though the exact size and order may vary. Described as a structure, a dictionary entry might look this way:[28]

```
  structure
    byte:       flag           \ 3bit flags + length of word's name
    char-array: name           \ name's runtime length isn't known at compile time
    address:    previous       \ link field, backward ptr to previous word
    address:    codeword       \ ptr to the code to execute this word
    any-array:  parameterfield \ unknown length of data, words, or opcodes
  end-structure forthword
```

The name field starts with a prefix giving the length of the word's name (typically up to 32 bytes), and several bits for flags. The character representation of the word's name then follows the prefix. Depending on the particular implementation of Forth, there may be one or more NUL ('\0') bytes for alignment.

The link field contains a pointer to the previously defined word. The pointer may be a relative displacement or an absolute address that points to the next oldest sibling.

The code field pointer will be either the address of the word which will execute the code or data in the parameter field or the beginning of machine code that the processor will execute directly. For colon defined words, the code field pointer points to the word that will save the current Forth instruction pointer (IP) on the return stack, and load the IP with the new address from which to continue execution of words. This is the same as what a processor's call/return instructions do.

Structure of the compiler

The compiler itself is not a monolithic program. It consists of Forth words visible to the system, and usable by a programmer. This allows a programmer to change the compiler's words for special purposes.

The "compile time" flag in the name field is set for words with "compile time" behavior. Most simple words execute the same code whether they are typed on a command line, or embedded in code. When compiling these, the compiler simply places code or a threaded pointer to the word.[22]

The classic examples of compile-time words are the control structures such as `IF` and `WHILE`. Almost all of Forth's control structures and almost all of its compiler are implemented as compile-time words. Apart from some rarely used control flow words only found in a few implementations, such as a conditional return, all of Forth's control flow words are executed during compilation to compile various combinations of primitive words along with their branch addresses. For instance, `IF` and `WHILE`, and the words that match with those, set up `BRANCH` (unconditional branch) and `?BRANCH` (pop a value off the stack, and branch if it is false). Counted loop control flow words work similarly but set up combinations of primitive words that work with a counter, and so on. During compilation, the data stack is used to support control structure balancing, nesting, and back-patching of branch addresses. The snippet:

```
  ... DUP 6 < IF DROP 5 ELSE 1 - THEN ...
```

would be compiled to the following sequence inside a definition:

```
  ... DUP LIT 6 < ?BRANCH 5  DROP LIT 5  BRANCH 3  LIT 1 - ...
```

The numbers after `BRANCH` represent relative jump addresses. `LIT` is the primitive word for pushing a "literal" number onto the data stack.

Compilation state and interpretation state

The word `:` (colon) parses a name as a parameter, creates a dictionary entry (a *colon definition*) and enters compilation state. The interpreter continues to read space-delimited words from the user input device. If a word is found, the interpreter executes the *compilation semantics* associated with the word, instead of the *interpretation semantics*. The default compilation semantics of a word are to append its interpretation semantics to the current definition.[22]

The word `;` (semi-colon) finishes the current definition and returns to interpretation state. It is an example of a word whose compilation semantics differ from the default. The interpretation semantics of `;` (semi-colon), most control flow words, and several other words are undefined in ANS Forth, meaning that they must only be used inside of definitions and not on the interactive command line.[22]

The interpreter state can be changed manually with the words `[` (left-bracket) and `]` (right-bracket) which enter interpretation state or compilation state, respectively. These words can be used with the word `LITERAL` to calculate a value during a compilation and to insert the calculated value into the current colon definition. `LITERAL` has the compilation semantics to take an object from the data stack and to append semantics to the current colon definition to place that object on the data stack.

In ANS Forth, the current state of the interpreter can be read from the flag `STATE` which contains the value true when in compilation state and false otherwise. This allows the implementation of so-called *state-smart words* with behavior that changes according to the current state of the interpreter.

Immediate words

The word `IMMEDIATE` marks the most recent colon definition as an *immediate word*, effectively replacing its compilation semantics with its interpretation semantics.[29] Immediate words are normally executed during compilation, not compiled, but this can be overridden by the programmer in either state. `;` is an example of an immediate word. In ANS Forth, the word `POSTPONE` takes a name as a parameter and appends the compilation semantics of the named word to the current definition even if the word was marked immediate. Forth-83 defined separate words `COMPILE` and `[COMPILE]` to force the compilation of non-immediate and immediate words, respectively.

Unnamed words and execution tokens

In ANS Forth, unnamed words can be defined with the word `:NONAME` which compiles the following words up to the next `;` (semi-colon) and leaves an *execution token* on the data stack. The execution token provides an opaque handle for the compiled semantics, similar to the function pointers of the C programming language.

Execution tokens can be stored in variables. The word `EXECUTE` takes an execution token from the data stack and performs the associated semantics. The word `COMPILE,` (compile-comma) takes an execution token from the data stack and appends the associated semantics to the current definition.

The word `'` (tick) takes the name of a word as a parameter and returns the execution token associated with that word on the data stack. In interpretation state, `' RANDOM-WORD EXECUTE` is equivalent to `RANDOM-WORD`.[30]

Parsing words and comments

The words `:` (colon), `POSTPONE`, `'` (tick) are examples of *parsing words* that take their arguments from the user input device instead of the data stack. Another example is the word `(` (paren) which reads and ignores the following words up to and including the next right parenthesis and is used to place comments in a colon definition. Similarly, the word `\` (backslash) is used for comments that continue to the end of the current line. To be parsed correctly, `(` (paren) and `\` (backslash) must be separated by whitespace from the following comment text.

Structure of code

In most Forth systems, the body of a code definition consists of either machine language, or some form of threaded code. The original Forth which follows the informal FIG standard (Forth Interest Group), is a TIL (Threaded Interpretive Language). This is also called indirect-threaded code, but direct-threaded and subroutine threaded Forths have also become popular in modern times. The fastest modern Forths, such as SwiftForth, VFX Forth, and iForth, compile Forth to native machine code.

Data objects

When a word is a variable or other data object, the CF points to the runtime code associated with the defining word that created it. A defining word has a characteristic "defining behavior" (creating a dictionary entry plus possibly allocating and initializing data space) and also specifies the behavior of an instance of the class of words constructed by this defining word. Examples include:

`VARIABLE`
: Names an uninitialized, one-cell memory location. Instance behavior of a `VARIABLE` returns its address on the stack.

`CONSTANT`
: Names a value (specified as an argument to `CONSTANT`). Instance behavior returns the value.

`CREATE`
: Names a location; space may be allocated at this location, or it can be set to contain a string or other initialized value. Instance behavior returns the address of the beginning of this space.

Forth also provides a facility by which a programmer can define new application-specific defining words, specifying both a custom defining behavior and instance behavior. Some examples include circular buffers, named bits on an I/O port, and automatically indexed arrays.

Data objects defined by these and similar words are global in scope. The function provided by local variables in other languages is provided by the data stack in Forth (although Forth also has real local variables). Forth programming style uses very few named data objects compared with other languages; typically such data objects are used to contain data which is used by a number of words or tasks (in a multitasked implementation).[31]

Forth does not enforce consistency of data type usage; it is the programmer's responsibility to use appropriate operators to fetch and store values or perform other operations on data.

Programming

Words written in Forth are compiled into an executable form. The classical "indirect threaded" implementations compile lists of addresses of words to be executed in turn; many modern systems generate actual machine code (including calls to some external words and code for others expanded in place). Some systems have optimizing compilers. Generally speaking, a Forth program is saved as the memory image of the compiled program with a single command (e.g., RUN) that is executed when the compiled version is loaded.

During development, the programmer uses the interpreter in REPL mode to execute and test each little piece as it is developed. Most Forth programmers therefore advocate a loose top-down design, and bottom-up development with continuous testing and integration.[32]

The top-down design is usually separation of the program into "vocabularies" that are then used as high-level sets of tools to write the final program. A well-designed Forth program reads like natural language, and implements not just a single solution, but also sets of tools to attack related problems.[33]

Code examples

"Hello, World!"

One possible implementation:

```
: HELLO ( -- ) CR ." Hello, World!" ;
```

```
HELLO <cr>
Hello, World!
```

The word `CR` (Carriage Return) causes the following output to be displayed on a new line. The parsing word `."` (dot-quote) reads a double-quote delimited string and appends code to the current definition so that the parsed string will be displayed on execution. The space character separating the word `."` from the string `Hello, World!` is not included as part of the string. It is needed so that the parser recognizes `."` as a Forth word.

A standard Forth system is also an interpreter, and the same output can be obtained by typing the following code fragment into the Forth console:

```
CR .( Hello, World!)
```

`.(` (dot-paren) is an immediate word that parses a parenthesis-delimited string and displays it. As with the word `."` the space character separating `.(` from `Hello, World!` is not part of the string.

The word `CR` comes before the text to print. By convention, the Forth interpreter does not start output on a new line. Also by convention, the interpreter waits for input at the end of the previous line, after an `ok` prompt. There is no implied "flush-buffer" action in Forth's `CR`, as sometimes is in other programming languages.

Mixing states of compiling and interpreting

Here is the definition of a word `EMIT-Q` which when executed emits the single character `Q`:

```
: EMIT-Q   81 ( the ASCII value for the character 'Q' ) EMIT ;
```

This definition was written to use the ASCII value of the `Q` character (81) directly. The text between the parentheses is a comment and is ignored by the compiler. The word `EMIT` takes a value from the data stack and displays the corresponding character.

The following redefinition of `EMIT-Q` uses the words `[` (left-bracket), `]` (right-bracket), `CHAR` and `LITERAL` to temporarily switch to interpreter state, calculate the ASCII value of the `Q` character, return to compilation state and append the calculated value to the current colon definition:

```
: EMIT-Q   [ CHAR Q ]  LITERAL  EMIT ;
```

The parsing word `CHAR` takes a space-delimited word as parameter and places the value of its first character on the data stack. The word `[CHAR]` is an immediate version of `CHAR`. Using `[CHAR]`, the example definition for `EMIT-Q` could be rewritten like this:

```
: EMIT-Q   [CHAR] Q  EMIT ; \ Emit the single character 'Q'
```

This definition used `\` (backslash) for the describing comment.

Both `CHAR` and `[CHAR]` are predefined in ANS Forth. Using `IMMEDIATE` and `POSTPONE`, `[CHAR]` could have been defined like this:

```
: [CHAR]   CHAR  POSTPONE LITERAL ; IMMEDIATE
```

A complete RC4 cipher program

In 1987, Ron Rivest developed the RC4 cipher-system for RSA Data Security, Inc. The code is extremely simple and can be written by most programmers from the description:

> We have an array of 256 bytes, all different. Every time the array is used it changes by swapping two bytes. The swaps are controlled by counters *i* and *j*, each initially 0. To get a new *i*, add 1. To get a new *j*, add the array byte at the new *i*. Exchange the array bytes at *i* and *j*. The code is the array byte at the sum of the array bytes at *i* and *j*. This is XORed with a byte of the plaintext to encrypt, or the ciphertext to decrypt. The array is initialized by first setting it to 0 through 255. Then step through it using *i* and *j*, getting the new *j* by adding to it the array byte at *i* and a key byte, and swapping the array bytes at *i* and *j*. Finally, *i* and *j* are set to 0. All additions are modulo 256.

The following Standard Forth version uses Core and Core Extension words only.

```
0 value ii         0 value jj
0 value KeyAddr    0 value KeyLen
create SArray    256 allot    \ state array of 256 bytes
: KeyArray       KeyLen mod   KeyAddr ;

: get_byte       + c@ ;
: set_byte       + c! ;
: as_byte        255 and ;
```

```
: reset_ij      0 TO ii   0 TO jj ;
: i_update      1 +   as_byte TO ii ;
: j_update      ii SArray get_byte +   as_byte TO jj ;
: swap_s_ij
    jj SArray get_byte
       ii SArray get_byte  jj SArray set_byte
    ii SArray set_byte
;

: rc4_init ( KeyAddr KeyLen -- )
    256 min TO KeyLen   TO KeyAddr
    256 0 DO   i i SArray set_byte   LOOP
    reset_ij
    BEGIN
        ii KeyArray get_byte   jj +  j_update
        swap_s_ij
        ii 255 < WHILE
        ii i_update
    REPEAT
    reset_ij
;
: rc4_byte
    ii i_update   jj j_update
    swap_s_ij
    ii SArray get_byte   jj SArray get_byte +   as_byte SArray get_byte  xor
;
```

This is one of many ways to test the code:

```
hex
create AKey   61 c, 8A c, 63 c, D2 c, FB c,
: test   cr   0 DO  rc4_byte . LOOP  cr ;
AKey 5 rc4_init
2C F9 4C EE DC  5 test   \ output should be: F1 38 29 C9 DE
```

Implementations

Because the Forth is simple to implement and has no standard reference implementation, there are numerous versions of the language. In addition to supporting the standard varieties of desktop computer systems (POSIX, Microsoft Windows, macOS), many of these Forth systems also target a variety of embedded systems. Listed here are some of the more systems which conform to the 1994 ANS Forth standard.

- Gforth, a portable ANS Forth implementation from the GNU Project
- SwiftForth (https://www.forth.com/), native code desktop and embedded Forths by Forth, Inc.
- VFX Forth, highly-optimizing native code Forth
- Win32Forth (https://sourceforge.net/projects/win32forth/), Windows-oriented, originally written in 1994-95 and continuously updated
- Open Firmware, a bootloader and Firmware standard based on ANS Forth
- pForth, portable Forth written in C
- SP-Forth, ANS Forth implementation from the Russian Forth Interest Group (RuFIG)
- Mecrisp-Stellaris (http://mecrisp.sourceforge.net), embedded Forth with introduction here hackaday-intro (https://hackaday.com/2017/01/27/forth-the-hackers-language/) with an easy demo on desktop here using docker demo (https://jeelabs.org/article/1720b/)

See also

- colorForth, a later, more minimal Forth-variant from Chuck Moore
- RTX2010, a CPU that runs Forth natively

References

1. NASA applications of Forth (https://web.archive.org/web/20101024223709/http://forth.gsfc.nasa.gov/) (original NASA server no longer running, copy from archive.org)
2. "Intersil's RTX processors and Forth software controlled the successful Philae landing" (http://www.mpeforth.com/press/MPE_PR_From_Telescope_to_Comet_2014_11_13.pdf) (PDF). *MicroProcessor Engineering Limited*. October 13, 2014.
3. "Here comes Philae! Powered by an RTX2010" (http://www.cpushack.com/2014/11/12/here-comes-philae-powered-by-an-rtx2010/). *The CPU Shack Museum*. October 12, 2014. Retrieved May 23, 2017.
4. Maher, Jimmy (October 28, 2014). "Starflight" (http://www.filfre.net/2014/10/starflight/). *The Digital Antiquarian*. Retrieved May 23, 2017.
5. Brodie, Leo. "Starting Forth" (https://www.forth.com/starting-forth/0-starting-forth/). *Forth dot com*. Forth, Inc. Retrieved July 14, 2020.
6. "Space Related Applications of Forth" (https://web.archive.org/web/20101024223709/http://forth.gsfc.nasa.gov/). Archived from the original (http://forth.gsfc.nasa.gov/) on 2010-10-24. Retrieved 2007-09-04.
7. C. H. Moore; E. D. Rather; D. R. Colburn (March 1993). "The Evolution of Forth" (http://www.forth.com/resources/evolution/index.html). *ACM SIGPLAN Notices*. ACM SIGPLAN History of Programming Languages. **28**.
8. "Atari In-Store Demonstration Program" (http://www.atarimania.com/demo-atari-400-800-xl-xe-in-store-demonstration-program_19329.html). *Atari Mania*.
9. Maynard, David S. "David Maynard: Software Artist" (https://www.software-artist.com).
10. Campbell et al, "Up and Running with Asyst 2.0", MacMillan Software Co., 1987
11. Moore, Charles H (1991). "Forth - The Early Years" (https://web.archive.org/web/20060615025259/http://www.colorforth.com/HOPL.html). Archived from the original (http://www.colorforth.com/HOPL.html) on 2006-06-15. Retrieved 2006-06-03.
12. "The Forth-79 Standard" (https://www.physics.wisc.edu/~lmaurer/forth/Forth-79.pdf) (PDF). Archived (https://web.archive.org/web/20190412142437/https://www.physics.wisc.edu/~lmaurer/forth/Forth-79.pdf) (PDF) from the original on 2019-04-12.
13. "The Forth-83 Standard" (http://forth.sourceforge.net/standard/fst83/).
14. "Programming Languages: Forth" (http://www.taygeta.com/forth/dpans.html). ANSI technical committee X3J14. 24 March 1994. Retrieved 2006-06-03.
15. "Standard Forth (ANSI INCITS 215-1994) Reference" (http://quartus.net/files/PalmOS/Forth/Docs/stdref.pdf) (PDF). Quartus Handheld Software. 13 September 2005. Retrieved 2013-04-14.
16. "The Forth Language" (https://archive.org/details/byte-magazine-1980-08/), *BYTE Magazine*, **5** (8), 1980
17. M. Anton Ertl. "Forth family tree and timeline" (http://www.complang.tuwien.ac.at/forth/family-tree/).
18. Lutus, Paul (1982). "GraFORTH Language Manual" (https://archive.org/details/graforth1). *archive.org*. Insoft.
19. Claunch, Carl (2018-03-02). "Restoring the original source code for FORTH on the IBM 1130" (https://rescue1130.blogspot.com/2018/03/restoring-original-source-code-for.html). *rescue1130*. Retrieved July 30, 2018.
20. Brodie, Leo (1987). *Starting Forth* (Second ed.). Prentice-Hall. p. 20. ISBN 978-0-13-843079-5.
21. Brodie, Leo (1987). *Starting Forth* (Second ed.). Prentice-Hall. p. 14. ISBN 978-0-13-843079-5.
22. Brodie, Leo (1987). *Starting Forth* (Second ed.). Prentice-Hall. p. 16. ISBN 978-0-13-843079-5.
23. Rodriguez, Brad. "B.Y.O.ASSEMBLER" (https://web.archive.org/web/20060623185833/http://www.zetetics.com/bj/papers/6809asm.txt). Archived from the original (http://www.zetetics.com/bj/papers/6809asm.txt) on 2006-06-23. Retrieved 2006-06-19.

24. Rodriguez, Brad. "MULTITASKING 8051 CAMELFORTH" (https://web.archive.org/web/20060622063041/http://www.zetetics.com/bj/papers/8051task.pdf) (PDF). Archived from the original (http://www.zetetics.com/bj/papers/8051task.pdf) (PDF) on 2006-06-22. Retrieved 2006-06-19.
25. Rodriguez, Brad (July 1995). "MOVING FORTH" (https://web.archive.org/web/20060623190109/http://www.zetetics.com/bj/papers/moving8.htm). Archived from the original (http://www.zetetics.com/bj/papers/moving8.htm) on 2006-06-23. Retrieved 2006-06-19.
26. Shoebridge, Peter (1998-12-21). "Motorola Background Debugging Mode Driver for Windows NT" (https://web.archive.org/web/20070606083244/http://www.zeecube.com/archive/bdm/index.htm). Archived from the original (http://www.zeecube.com/archive/bdm/index.htm) on 2007-06-06. Retrieved 2006-06-19.
27. Martin, Harold M. (March 1991). "Developing a tethered Forth model". *ACM Sigforth Newsletter*. ACM Press. **2** (3): 17–19. doi:10.1145/122089.122091 (https://doi.org/10.1145%2F122089.122091). S2CID 26362015 (https://api.semanticscholar.org/CorpusID:26362015).
28. Brodie, Leo (1987). *Starting Forth* (http://www.forth.com/starting-forth/) (Second ed.). Prentice-Hall. pp. 200–202. ISBN 978-0-13-843079-5.
29. Brodie, Leo (1987). *Starting Forth* (Second ed.). Prentice-Hall. p. 273. ISBN 978-0-13-843079-5.
30. Brodie, Leo (1987). *Starting Forth* (Second ed.). Prentice-Hall. p. 199. ISBN 978-0-13-843079-5.
31. Brodie, Leo (1987). "Under The Hood". *Starting Forth* (2nd ed.). Prentice-Hall. p. 241. ISBN 978-0-13-843079-5. "To summarize, there are three kinds of variables: System variables contain values used by the entire Forth system. User variables contain values that are unique for each task, even though the definitions can be used by all tasks in the system. Regular variables can be accessible either system-wide or within a single task only, depending upon whether they are defined within OPERATOR or within a private task."
32. Brodie, Leo (1984). *Thinking Forth* (https://archive.org/details/thinkingforthlan00brod). Prentice-Hall. ISBN 978-0-13-917568-8.
33. The classic washing machine example (http://www.forth.com/embedded/swiftx-embedded-systems-7.html) describes the process of creating a vocabulary to naturally represent the problem domain in a readable way.

Further reading

- Biancuzzi, Federico; Shane Warden (2009). "Chapter Four [A conversation with Chuck Moore]". *Masterminds of Programming, Conversations with the Creators of Major Programming Languages*. O'REILLY. ISBN 978-0-596-51517-1.
- Brodie, Leo (2007). Marcel Hendrix (ed.). *Starting Forth* (http://www.forth.com/starting-forth/index.html). Marlin Ouverson (Web ed.). FORTH, Inc. Retrieved 2007-09-29.
- Brodie, Leo (2004). Bernd Paysan (ed.). *Thinking Forth* (http://thinking-forth.sourceforge.net) (PDF Online book). ISBN 978-0-9764587-0-8. Retrieved 2008-09-15.
- Conklin, Edward K.; Elizabeth D. Rather; et al. (8 September 2007). *Forth Programmer's Handbook* (http://www.forth.com/forth/forth-books.html) (paperback) (3rd ed.). BookSurge Publishing. p. 274. ISBN 978-1-4196-7549-2.
- Rather, Elizabeth D. (2000). *Forth Application Techniques* (http://www.forth.com/forth/forth-books.html) (spiral bound). Forth Inc. p. 158. ISBN 978-0-9662156-1-8.
- Pelc, Stephen F. (2005). *Programming Forth* (http://www.mpeforth.com/books.htm) (spiral bound). MicroProcessor Engineering Ltd. p. 188.
- Kelly, Mahlon G.; Nicholas Spies (1986). *FORTH: A Text and Reference* (https://archive.org/details/forthtextreferen0000kell). Prentice-Hall. ISBN 978-0-13-326331-2.
- Koopman, Jr, Philip J. (1989). *Stack Computers: The New Wave* (http://www.ece.cmu.edu/~koopman/stack_computers/index.html) (hardcover). Ellis Horwood Limited. ISBN 978-0-7458-

0418-7.
- Pountain, Dick (1987). *Object-oriented Forth: Implementation of Data Structures*. Harcourt Brace Jovanovich. ISBN 978-0-12-563570-7.
- Payne, William (19 December 1990). *Embedded Controller Forth for the 8051 Family*. Elsevier. p. 528. ISBN 978-0-12-547570-9.
- Winfield, Alan (1983). *The Complete Forth* (https://archive.org/details/completeforth0000winf). John Wiley. ISBN 978-0471882350.
- Baglioni, Gio Federico (1983). *Forth per VIC20 e CBM64*. Jackson. ISBN 978-88-7056-141-8.
